

C# Básico

Programação Visual

Sumário

- História da Linguagem C#
- C# - Conceitos Básicos
- C# - Valores x Referências
- C# - Programação Orientada por Objetos
- C# - Características Herdadas do C++
- C# - Propriedades e Indexers

História da Linguagem C#

Objetivos do design, evolução da linguagem

Linguagem C# - Objetivos de design



- **Linguagem C# - objetivos de design:**

- Pretende-se que seja uma linguagem **orientada por objetos** simples, moderna e de uso geral,
- A linguagem e a sua implementação, devem **suportar os princípios de Engenharia de Software** tais como verificação de tipos forte, verificação de limites de arrays, deteção de variáveis que não foram inicializadas e *garbage collection* automática. São consideradas importantes a robustez, durabilidade e a produtividade.
- Pretende-se que seja usada para o **desenvolvimento de componentes** de software adequados à sua instalação em ambientes distribuídos.
- A **portabilidade** é muito importante para o código fonte e para programadores, especialmente os que estão familiarizados com as linguagens C e C++.
- O suporte para a **internacionalização** é muito importante.
- Pretende-se que o C# seja adequado para a **escrita de aplicações em sistemas anfitrião e embutidos**, desde os grandes sistemas que usam sistemas operativos sofisticados até aos pequenos sistemas com funções específicas.
- Apesar das aplicações C# serem projetadas para serem económicas no que diz respeito à memória e ao poder de processamento, não se pretende que a linguagem compita diretamente, em performance ou em tamanho, com as linguagens C e Assembly.



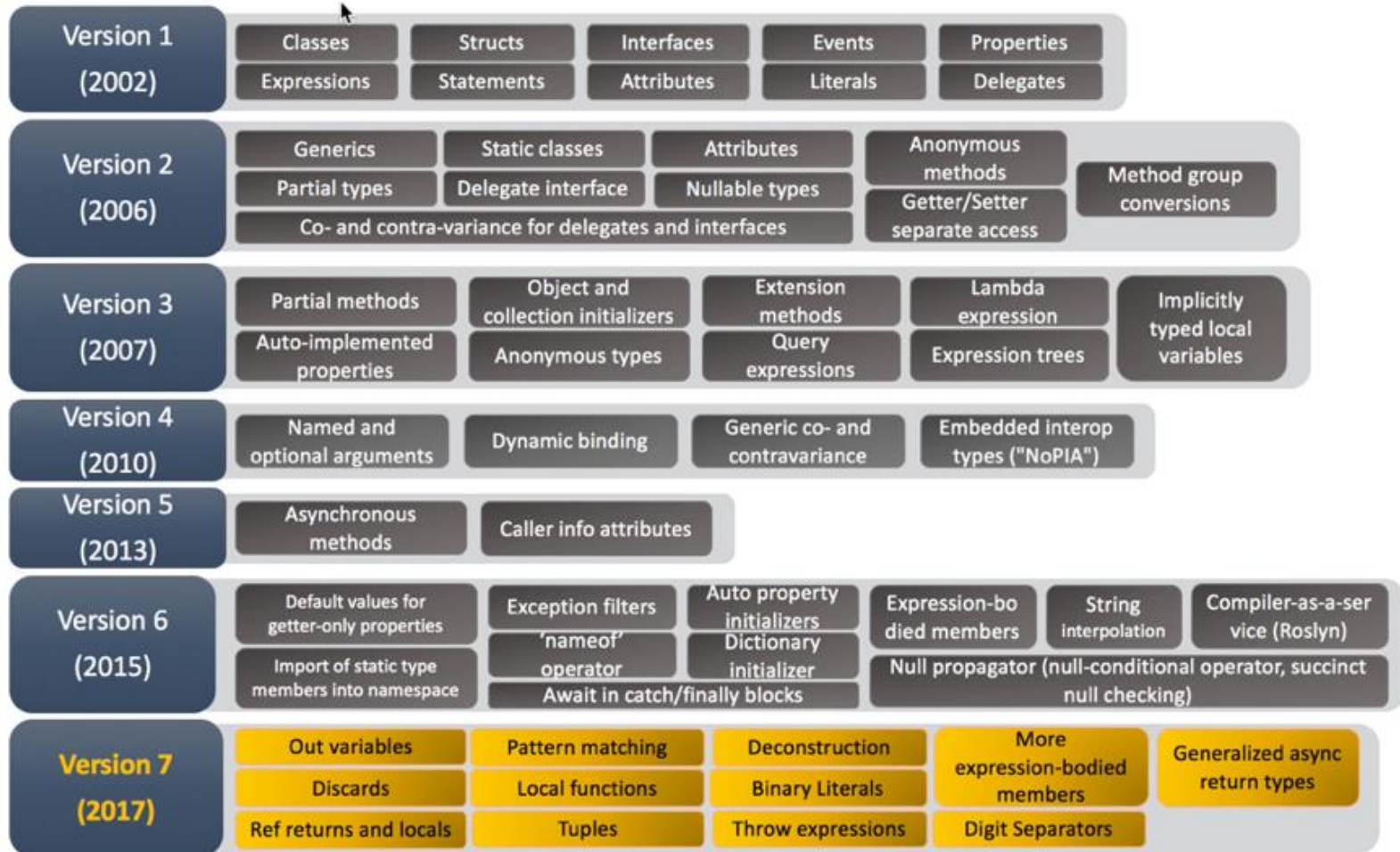
Evolução da linguagem C#



Versão	Especificação			Data	.Net Framework	Visual Studio
	Ecma	ISO/IEC	Microsoft			
C# 1.0	dez/02	abr/03	jan/02	jan/02	.Net Framework 1.0	Visual Studio .NET 2002
C# 1.1			out/03	abr/03	.Net Framework 1.1	Visual Studio .NET 2003
C# 1.2						
C# 2.0	jun/06	set/06	set/05	nov/05	.Net Framework 2.0	Visual Studio 2005
C# 3.0			ago/07	nov/07	.Net Framework 2.0 (exceto LINQ) .Net Framework 3.0 (exceto LINQ) .Net Framework 3.5	Visual Studio 2008 Visual Studio 2010
C# 4.0			abr/10	abr/10	.Net Framework 4	Visual Studio 2010
C# 5.0	dez/17	dez/18	jun/13	ago/12	.Net Framework 4.5	Visual Studio 2012 Visual Studio 2013
C# 6.0			Esboço	jul/15	.Net Framework 4.6	Visual Studio 2015
C# 7.0				mar/17	.Net Framework 4.6.2	Visual Studio 2017
C# 7.1				ago/17	.Net Framework 4.7, .NET Core 2.0	Visual Studio 2017 v15.3
C# 7.2				nov/17	.Net Framework 4.7.1, .NET Core 2.0	Visual Studio 2017 v15.5
C# 7.3				mai/18	.Net Framework 4.7.2, .NET Core 2.1	Visual Studio 2017 v15.7
C# 8.0				set/19	.Net Framework 4.8, .NET Core 3.0	Visual Studio 2019 v16.3
C# 9.0				set/20	.Net 5.0	Visual Studio 2019 v16.8
C# 10.0				Nov/21	.NET 6.0, .NET 6.01	Visual Studio 2022 v17.0
C# 11.0				Nov/22 ?	.NET 7.0 ??	



Evolução da linguagem C#



Evolução da linguagem C#



Preview

Version 7.1
(2017)

async Main method default literal expressions Inferred tuple element names
Pattern matching on generic type parameters

Version 7.2
(2017)

Leading underscores in numeric literals Non-trailing named arguments private protected access modifier
Techniques for writing safe efficient code Conditional ref expressions

Version 7.3
(2018)

new features to support the theme of better performance for safe code new compiler options: publicsign, pathmap
incremental improvements to existing features.

Version 8.0
(2019)

Readonly members Using declarations Nullable reference types
Disposable ref structs Asynchronous streams Stackalloc in nested expressions Enhanced of interpolated verbatim strings
Indices and ranges Default interface methods Unmanaged constructed types
Static local functions Null-coalescing assignment Pattern matching enhancements

Version 9.0
(2020)

Records Fit and finish features Support for code generators
Init only setters Top-level statements Performance and interop

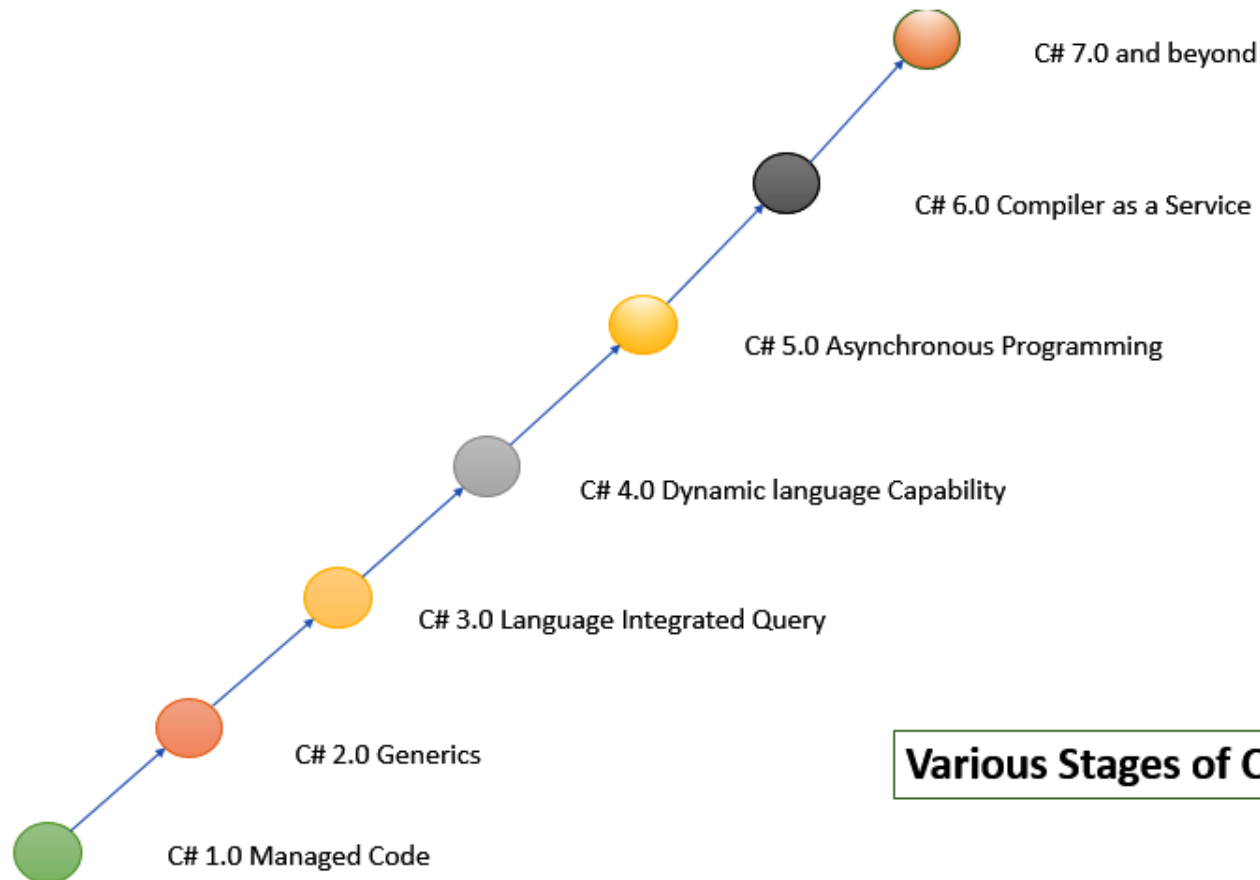
Version 10.0
(2021)

Record structs Improvements of structure types Interpolated string handler File scoped namespace declaration Record types can seal ToString Assignment and declaration in same deconstruction
Global using directives Extended property patterns Lambda expression improvements Constant interpolated strings
Improved definite assignment Allow AsyncMethodBuilder attribute on methods CallerArgumentExpression attribute diagnostics Enhanced #line pragma

Version 11.0
(2022)

Generic attributes Generic math support Numeric IntPtr and UIntPtr Newlines in string interpolations Improved method group conversion to delegate Pattern match Span<char> or ReadOnlySpan<char> on a constant string
List patterns Raw string literals Auto-default struct Extended nameof scope
UTF-8 string literals Required members ref fields and ref scoped variables File scoped types

Evolução da linguagem C#



Various Stages of C# language



C# - Conceitos Básicos

Programas simples, tipos de dados, operadores, estruturas de control e tabelas.



```
using System;

namespace HelloWorld
{
    public class Program
    {
        public static void Main(string[] args)
        {
            Console.WriteLine("Olá Mundo!");
            Console.ReadKey();
        }
    }
}
```



C# versus Java - Estrutura dos programas



C#

```
using System;

namespace HelloWorld
{

    /// <summary>
    /// Olá Mundo
    /// </summary>
    public class Program
    {
        public static void Main(string[] args)
        {
            Console.WriteLine("Olá Mundo!");
        }
    }
}
```

Java

```
package MyProgram;

import java.lang.System;

/**
 *
 * @author Jose Cordeiro
 */
public class Program {

    public static void main(String[] args) {

        System.out.println("Olá Mundo! ");

    }
}
```





Convenções de escrita do código

- Notação **PascalCase** para **identificadores públicos**
 - **Métodos**, classes, namespaces, **propriedades**, **valores dos tipos enumerados**, **constantes**
- Notação **camelCase** para **identificadores privados**
 - Atributos, parâmetros, variáveis locais
- **Interfaces** começam por '**I**' e seguem a notação **PascalCase**
 - Ex: ICollection, IEnumerable, IList
- Não usar '_' ou '-' nos identificadores
 - Nota: É comum usar o prefixo '_' nos atributos. Ex: _name, _price



C# - Tipos de dados predefinidos



Keyword	.NET Class	Data Type	Literal
byte	Byte	Unsigned integer	
sbyte	SByte	Signed integer	
int	Int32	Signed integer	1_000_000
uint	UInt32	Unsigned integer	13u
short	Int16	Signed integer	
ushort	UInt16	Unsigned integer	456u1
long	Int64	Signed integer	1231
ulong	UInt64	Unsigned integer	
float	Single	Single-precision floating point type	0.5f
double	Double	Double-precision floating point type	
char	Char	A single Unicode character	'c' 'X' '\t'
bool	Boolean	Logical Boolean type	True false
object	Object	Base type of all other types	
string	String	A sequence of characters	"Text example"
decimal	Decimal	Precise fractional or integral type that can represent decimal numbers with 29 significant digits	10.25m





C#

```
public sealed class FinalCircle
{
    public const double radius = 1.5;

    private readonly int id = 1;
}
```

Java

```
public final class FinalCircle {

    public static final double radius = 1.5;

    private final int id = 1;
}
```

- Em Java a palavra **final** tem diferentes utilizações. Em **C#**, existe uma palavra diferente para cada uma das formas de utilização:
 - Constante de classe: **const**
 - Constante de instância: **readonly**
 - Classe final (não é possível a herança): **sealed**

C# - Operadores



Operator category	Operators
Arithmetic	+ - * / %
Logical	! && true false
String concatenation	+
Increment, decrement	++ --
Bitwise	& ^ ~ << >>
Relational	== != < > <= >=
Assignment	= += -= *= /= %= &= = ^= <<= >>= ??
Member access	.
Indexing	[]
Cast	()
Conditional	?:
Delegate concatenation and removal	+ -
Object creation	new
Type information	as is sizeof typeof
Overflow exception control	checked unchecked
Indirection and Address	* -> [] &





```
string str = Console.ReadLine();
switch (str)
{
    case "janeiro":
    case "março":
    case "maio":
    case "julho":
    case "agosto":
    case "outubro":
    case "dezembro":
        Console.WriteLine("Tem 31 dias");
        break;
    case "abril":
    case "junho":
    case "setembro":
    case "novembro":
        Console.WriteLine("Tem {0} dias", 30);
        break;
    default:
        Console.WriteLine("Mês desconhecido");
        break;
    case "fevereiro":
        Console.WriteLine("Tem {0} ou {1} dias", 28, 29);
        break;
}
```

O **break** é obrigatório
para todos os **case**





- do-while, while, for – idênticos em C# e Java
- Foreach:

C#

```
int[] numbers = new int[] { 2, 7, 12, 9 };
```

```
foreach (int val in numbers)
{
    Console.WriteLine(val);
}
```

Java

```
Integer[] numbers = new Integer[]{2, 7, 12, 9};
```

```
for (int val : numbers) {
    System.out.println(val);
}
```



Os parenteses aparecem
sempre a seguir ao tipo

- **Unidimensionais**

- EX:

```
double[] vals = new double[10];  
vals[2] = 5.0;
```

- **Multidimensionais**

- EX:

```
int[,] vals = new int[10 , 4];  
vals[2,3] = 5;
```

- **Tabela de tabelas**

- EX:

```
byte[][] vals = new byte[5][];  
vals[1] = new byte[5];  
vals[1][0] = 2;
```



C# - Valores x Referências

Tipos por valor e referência. Passagem de parâmetros por valor e por referência.

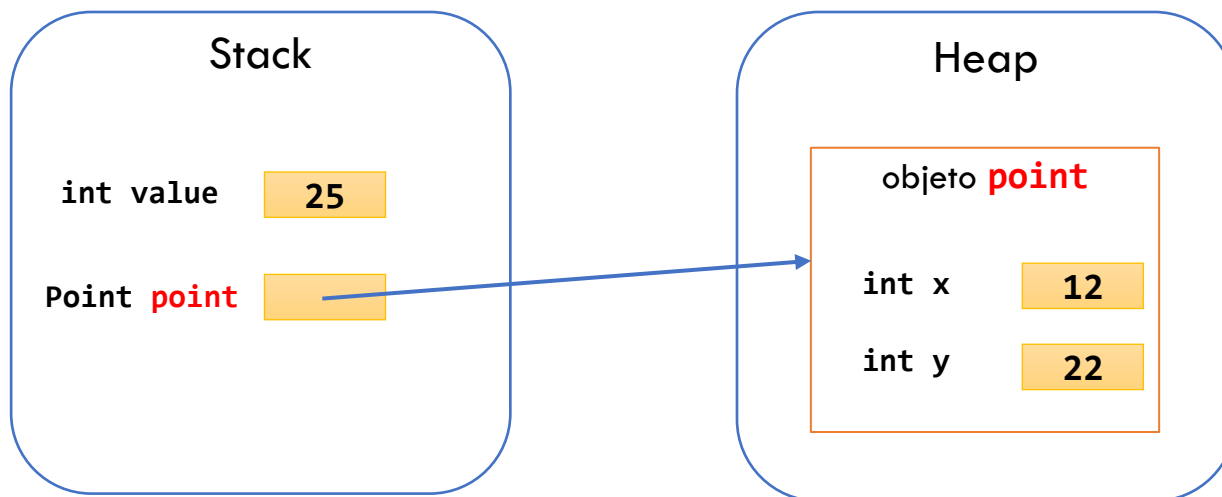


- **Tipos por valor**

- As variáveis destes tipos guardam um valor (no stack)
 - EX: `int`, `char`, `long`, `struct`

- **Tipos por referência**

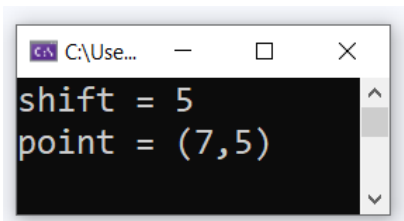
- As variáveis destes tipos guardam uma referência para o objeto (no heap)
 - EX: qualquer variável de uma classe, arrays.





- **Passagem por valor** – É o caso habitual e é feita por cópia dos valores passados. No caso de se passar:
 1. **Um valor:** o parâmetro recebe uma cópia desse valor. Qualquer alteração ao parâmetro não afeta o valor que estava na variável passada.

```
Point point = new Point(2, 5);  
int shift = 5;  
  
point.MoveX(shift);  
  
Console.WriteLine("shift = " + shift);  
Console.WriteLine("point = " + point);
```



```
public class Point  
{  
    private int x, y;  
  
    public Point(int x, int y)  
    {  
        this.x = x;  
        this.y = y;  
    }  
  
    public void MoveX(int value)  
    {  
        value += x;  
        x = value;  
    }  
  
    public override string ToString()  
    {  
        return $"({x},{y})";  
    }  
}
```



- **Passagem por valor** – É o caso habitual e é feita por cópia dos valores passados. No caso de se passar:

2. **Uma referência:** o parâmetro recebe uma cópia da referência passada. Alterações ao objeto referenciado afetam o objeto que estava na referência passada. As alterações à referência não afetam a referência que foi passada.

```
Point point1 = new Point(1, -2);
Point point2 = new Point(2, 5);

Console.WriteLine("point1 = " + point1);
Console.WriteLine("point2 = " + point2);
Console.WriteLine();

point1.SwapWith(point2);

Console.WriteLine("point1 = " + point1);
Console.WriteLine("point2 = " + point2);
```

```
C:\Users\Jose-...
point1 = (1,-2)
point2 = (2,5)

point1 = (2,5)
point2 = (1,-2)
```

```
public class Point
{
    private int x, y;

    public Point(int x, int y)
    {
        this.x = x;
        this.y = y;
    }

    public void SwapWith(Point point)
    {
        int tempX = point.x;
        int tempY = point.y;

        point.x = x;
        point.y = y;

        x = tempX;
        y = tempY;
    }

    public override string ToString()
    {
        return $"({x},{y})";
    }
}
```



C# - Métodos: passagem de parâmetros



- **Passagem por referência** – Possível em C# utilizando a keyword **ref** ou a keyword **out** antes do tipo do parâmetro. Neste caso, os parâmetros passam a representar 'alias' das variáveis passadas e as alterações feitas dentro do método funcionam como se se estivesse a alterar essas variáveis

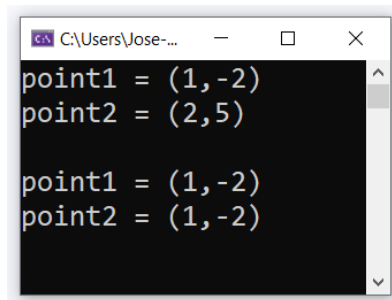
Nota: **ref** e **out** têm de ser usados também na chamada dos métodos antes das variáveis que afectam.

```
Point point1 = new Point(1, -2);
Point point2 = new Point(2, 5);

Console.WriteLine("point1 = " + point1);
Console.WriteLine("point2 = " + point2);
Console.WriteLine();

point1.ExportTo(ref point2);

Console.WriteLine("point1 = " + point1);
Console.WriteLine("point2 = " + point2);
```



```
C:\Users\Jose-...
point1 = (1,-2)
point2 = (2,5)

point1 = (1,-2)
point2 = (1,-2)
```

```
public class Point
{
    private int x, y;

    public Point(int x, int y)
    {
        this.x = x;
        this.y = y;
    }

    public void ExportTo(ref Point point)
    {
        point = new Point(x, y);
    }

    public override string ToString()
    {
        return $"({x},{y})";
    }
}
```



C# - Programação Orientada por Objetos

Campos, métodos, construtores. Paâmetros dos métodos.
Herança, polimorfismo, interfaces e classes abstratas.



- **Atributos** – chamados em C# de ‘campos’
 - Podem ser inicializados na declaração.
- **Métodos**
 - Código definido dentro da classe
 - Podem ter várias formas com o mesmo identificador.
- **Construtores**
 - O construtor por omissão inicializa todos os ‘campos’
 - Podem existir outros construtores, podendo ser então necessário definir o construtor sem argumentos.

C# versus Java – Métodos



- Métodos: Número variável de parâmetros
 - Em Java e C# são usados *arrays* para guardar os valores.

C#

```
public class Drawing
{
    // ...
    public void Add(params Figure[] figs)
    {
        foreach (Figure fig in figs)
        {
            figures.Add(fig);
        }
    }
}
```

// Example

```
Drawing draw = new Drawing();
```

```
draw.Add(new Circle());
```

```
draw.Add(new Circle(), new Square());
```

Java

```
public class Drawing {

    // ...
    public void add(Figure... figs) {

        for(Figure fig : figs) {

            figures.add(fig);
        }
    }
}
```

// Example

```
Drawing draw = new Drawing();
```

```
draw.add(new Circle());
```

```
draw.add(new Circle(), new Square());
```





- Métodos: Parâmetros opcionais e parâmetros nomeados

Parâmetros opcionais

```
public class Order
{
    public void Buy(int orderNumber,
                   string product = "unknown",
                   int quantity = 1)
    {
        // ...
    }
}

// Call:

Order order = new Order();

order.Buy(11); // product="unknown", quantity=1
order.Buy(17, "Pencil"); // quantity=1
order.Buy(25, "Pen", 12);
```

Parâmetros nomeados

```
public class Order
{
    public void PrintDetails(string seller,
                           int order,
                           string product)
    {
        // ...
    }
}

// Call

Order order = new Order();

order.PrintDetails("John", 3, "Mug");
order.PrintDetails(order:3, product:"Mug", seller:"John");
order.PrintDetails("John", 3, product: "Mug");
```



C# versus Java - Classes



C#

```
public class Figure
{
    private int x, y;

    public Figure() : this(0, 0)
    {
    }

    public Figure(int x, int y)
    {
        this.x = x;
        this.y = y;
    }

    public virtual string GetValues()
    {
        return "(" + x + "," + y + ")";
    }
}
```

this em posições diferentes

Java

```
public class Figure {

    private int x, y;

    public Figure() {
        this(0, 0);
    }

    public Figure(int x, int y) {
        this.x = x;
        this.y = y;
    }

    public String getValues() {
        return "(" + x + "," + y + ")";
    }
}
```



C# versus Java - Herança



C#

```
public class Circle : Figure
{
    private int radius = 1;

    public Circle(int x, int y, int radius) : base(x, y)
    {
        this.radius = radius;
    }

    public override string GetValues()
    {
        return base.GetValues() +
            "r=" + radius;
    }
}
```

base em vez de super e em posições diferentes

Java

```
public class Circle extends Figure {

    private int radius = 1;

    public Circle(int x, int y, int radius) {
        super(x, y);
        this.radius = radius;
    }

    @Override
    public String getValues() {
        return super.getValues()
            + "r=" + radius;
    }
}
```



C# versus Java - Herança



```
public class Figure
{
    private int x, y;

    public Figure(int x, int y)
    {
        this.x = x;
        this.y = y; }

    public virtual string GetValues()
    {
        return "(" + x + "," + y + ")";
    }
}
```

```
public class Circle : Figure
{
    private int radius = 1;

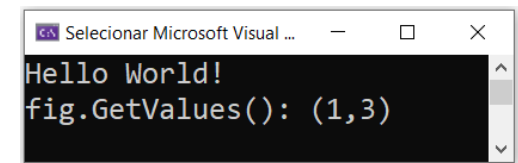
    public Circle(int x, int y, int radius) : base(x, y)
    {
        this.radius = radius;
    }

    public override string GetValues()
    {
        return base.GetValues() +
            "r=" + radius;
    }
}
```

Figure fig = new Circle(1, 3, 5);

Console.WriteLine("fig.GetValues(): " + fig.GetValues());

Sem **virtual** e **override**:



C# versus Java - Herança



C#

```
public class Circle : Figure
{
    private int radius = 1;

    public Circle(int x, int y, int radius) : base(x, y)
    {
        this.radius = radius;
    }

    public override string GetValues()
    {
        return base.GetValues() +
            "r=" + radius;
    }
}
```

Java

```
public class Circle extends Figure {

    private int radius = 1;

    public Circle(int x, int y, int radius) {
        super(x, y);
        this.radius = radius;
    }

    public String getValues() {
        return super.getValues()
            + "r=" + radius;
    }
}
```





- Classe Abstrata

- Usa a keyword '**abstract**'
- Não é possível a criação de objetos dessa classe
- Pode ter métodos abstratos (sem implementação)
 - Usam o modificador abstract
 - Possuem apenas a declaração
 - São automaticamente **virtuais**
- Pode ter métodos concretos
- Define uma **classe incompleta**



- Interfaces

- Por convenção os identificadores começam pela letra I
- Apenas declaram métodos, sem campos.
 - Não permitem declarar variáveis como em Java
- Todos os seus métodos são públicos e virtuais
- Podem incluir a declaração de propriedades.

C# - Características Herdadas do C++

Estruturas e redefinição de operadores.



- Estruturas em **C#**
 - São tipos por valor
 - Não é então necessário usar **new** na criação do valor.
 - São semelhantes às classes, mas:
 - Não têm um constructor por omissão.
 - Não têm herança.
 - Os campos são normalmente públicos.

```
public struct Point
{
    public int x;
    public int y;

    public Point(int x, int y)
    {
        this.x = x;
        this.y = y;
    }

    public void Print()
    {
        Console.WriteLine("x = {0}, y = {1}", x, y);
    }
}
```



- Redefinição de operadores **C#**
 - Possível em C#
 - Nem todos os operadores são redefiníveis

```
public class Complex
{
    private double imag, real;

    public Complex(double real, double imaginary = 0.0)
    {
        imag = imaginary;
        this.real = real;
    }

    public static Complex operator +(Complex a, Complex b)
    {
        return new Complex(a.real + b.real, a.imag + b.imag);
    }
}
```

// Example

```
Complex complex1 = new Complex(1.0, -2.0);
Complex complex2 = new Complex(5.0);
```

```
Complex complex3 = complex1 + complex2;
```



C# - Propriedades e Indexers

Propriedades explícitas e implícitas, indexers.



C#

```
public class Circle
{
    private int radius = 1;

    public int Radius
    {
        get
        {
            return radius;
        }

        set
        {
            this.radius = value;
        }
    }
}

// Example:

Circle c = new Circle();

c.Radius = 10;

Console.WriteLine( "Radius=" + c.Radius );
```

Java

```
public class Circle {

    private int radius = 1;

    public int getRadius() {
        return radius;
    }

    public void setRadius(int value) {
        this.radius = value;
    }
}

// Example:

Circle c = new Circle();

c.setRadius(10);

System.out.println("Radius=" + c.getRadius() );
```



C# - Propriedades explícitas e implícitas



Explícitas

```
public class Circle
{
    private int radius = 1;

    public int Radius
    {
        get
        {
            return radius;
        }

        set
        {
            this.radius = value;
        }
    }
}

// Example:

Circle c = new Circle();

c.Radius = 10;

Console.WriteLine( "Radius=" + c.Radius );
```

Implícitas

```
public class Circle
{
    public int Radius { get; set; } = 1;
}

// Nota: o campo continua a existir
//      mas não temos acesso direto a ele

// Example:

Circle c = new Circle();

c.Radius = 10;

Console.WriteLine( "Radius=" + c.Radius );
```





- Indexers em **C#**
 - Usar a notação de arrays com classes normais
 - O índice pode ser de qualquer tipo e pode existir mais do que um índice.

```
public class Drawing
{
    IList<Figure> figures = new List<Figure>();

    public void Add(params Figure[] figs)
    {
        foreach (Figure fig in figs)
        {
            figures.Add(fig);
        }
    }

    public Figure this[int index]
    {
        get
        {
            return figures[index];
        }
        set
        {
            figures[index] = value;
        }
    }
}
```

// Example

```
Drawing drawing = new Drawing();
drawing.Add( new Circle(10, 15, 4));
```

```
Console.WriteLine( "Figure 0" + drawing[0] );
Drawing[0] = new Circle(5, 5, 2));
```



Referências

- [https://en.wikipedia.org/wiki/C_Sharp_\(programming_language\)](https://en.wikipedia.org/wiki/C_Sharp_(programming_language))
- <https://docs.microsoft.com/en-us/dotnet/csharp/whats-new/csharp-version-history>
- <https://www.strive2code.com/post/2017/09/21/c-on-steroids-or-what-s-new-in-c-7-0>
- <https://www.packtpub.com/product/hands-on-object-oriented-programming-with-c/9781788296229>